Amortized Resource Analysis with Polymorphic Recursion and Partial Big-Step Operational Semantics

Jan Hoffmann* and Martin Hofmann

Ludwig-Maximilians-Universität München

Abstract. This paper studies the problem of statically determining upper bounds on the resource consumption of first-order functional programs. A previous work approached the problem with an automatic type-based amortized analysis for polynomial resource bounds. The analysis is parametric in the resource and can be instantiated to heap space, stack space, or clock cycles. Experiments with a prototype implementation have shown that programs are analyzed efficiently and that the computed bounds exactly match the measured worst-case resource behavior for many functions. This paper describes the inference algorithm that is used in the implementation of the system. It can deal with resource-polymorphic recursion which is required in the type derivation of many functions. The computation of the bounds is fully automatic if a maximal degree of the polynomials is given. The soundness of the inference is proved with respect to a novel operational semantics for partial evaluations to show that the inferred bounds hold for terminating as well as non-terminating computations. A corollary is that run-time bounds also establish the termination of programs.

1 Introduction

The quantitative analysis of algorithms is a classic problem in computer science. For many applications in software development it is necessary to obtain not only asymptotic bounds but rather specific upper bounds for concrete implementations. This is especially the case for the development of embedded and safety-critical systems.

Even for basic programs, manual analysis of the specific (non-asympt.) costs is tedious and error-prone. The problem gets increasingly complex for high-level programming languages, since one needs to be aware of the translation performed by the compiler. As a result, automatic methods for analyzing the resource behavior of programs have been the subject of extensive research (see §7).

Our approach to the problem follows a line of research that was initiated by Hofmann and Jost [1]. It is based on the potential method of amortized analysis that has been invented by Sleator and Tarjan [2] to simplify the manual reasoning about the costs of a sequence of operations that manipulate a data structure. [1] showed that a fully automatic amortized resource analysis can efficiently compute bounds on the heap-space consumption of many (first-order) functional programs that admit *linear* resource bounds. The limitation to linear bounds and accordingly linear constraints was essential for the efficiency of the analysis. Subsequent research considerably extended the range

* Supported by the DFG Graduiertenkolleg 1480 (PUMA).

of type-based amortized analysis, but the restriction to linear bounds remained. Examples are the extensions of type-based amortized analysis to object-oriented programs [3, 4], to generic resource metrics [5, 6], to polymorphic and higher-order programs [7], and to Java-like bytecode by means of separation logic [8].

Somewhat unexpectedly, we recently discovered a technique [9] that yields an automatic amortized analysis for polynomial bounds while still relying on linear constraint solving only. The resulting system efficiently computes resource bounds for first-order functional programs that admit bounds that are sums $\sum p_i(n_i)$ of univariate polynomials p_i. This includes bounds on the heap-space usage and the number of evaluation steps for a number of interesting functions such as quick sort, merge sort, insertion sort, longest common subsequence via dynamic programming, breadth-first traversal of a tree using a functional queue, and sieve of Eratosthenes.

The system has been implemented for *Resource Aware ML (RAML)* which is a first-order fragment of OCAML. It is available online[1] and can be run in a web browser to analyze example programs and user-generated code. Our experiments show that the computed bounds exactly match the measured worst-case behavior in many cases. For example we obtain tight evaluation-step bounds for quick sort and insertion sort.

The basic idea of the analysis is to fix a maximal degree k and then to collect linear constraints on the coefficients of polynomials of this degree. One can iteratively increase the degree so as to avoid costly computations earlier on. A fine point arises from the fact that polynomials must be nonnegative and monotone and that in order for allowing local constraint generation for pattern matches the class of allowed polynomials must be closed under the operation $p(n) \mapsto p(n+1) - p(n)$. This naturally leads to nonnegative linear combinations of binomial coefficients.

A further challenge for the inference of polynomial bounds is the need to deal with *resource-polymorphic recursion* (see §2), which is required to type most of the above example programs. However, it seems to be a hard problem to infer general resource polymorphic recursion even for the original linear system.

In this paper we present a pragmatic approach to resource-polymorphic recursion that works well and efficiently in practice. Despite being not complete with respect to the type rules, it infers types for most functions that admit a type-derivation, including the above examples. A somewhat artificial function that admits a resource-polymorphic typing that cannot be inferred by our algorithm is given in the extended version.

The main theorem of the paper (see §5) shows that the resource bounds are sound with respect to a big-step operational semantics. A dissatisfying feature of classical big-step semantics is that it does not provide evaluation judgments for non-terminating evaluations. As a result, the soundness theorems for amortized resource analyses have in the past been formulated for terminating evaluations only [1, 5, 7].

A secondary contribution of this paper is the introduction of a novel big-step operational semantics for partial evaluations which agrees with the usual big-step semantics on terminating computations. In this way, we retain the advantages of big-step semantics (shorter, less syntactic proofs; better agreement (arguably) with actual behaviour of computers) while capturing the resource behaviour of non-terminating programs. This enables the proof of an improved soundness result: if the type analysis has established a

[1] See `http://raml.tcs.ifi.lmu.de`.

resource bound then the resource consumption of the (possibly non-terminating) evaluation does not exceed the bound. It follows that run-time bounds also ensure termination.

This paper complements a previous paper [9]. The main contributions are as follows. We introduce a novel operational semantics for partial evaluations that allows a simplified and improved soundness theorem (in §4). We present algorithmic typing rules used by the inference algorithm (in §5). An extended soundness proof shows that the inferred bounds hold for both terminating and non-terminating computations (Thm. 4). We describe an inference algorithm that efficiently computes resource-polymorphic types for most functions for which such a type exists (in §6).

An extended version of this paper is available on the first author's website. It contains proofs, a case study on sorting algorithms in RAML, and a summary of our experiments with the inference algorithm.

2 Informal Presentation

Linear Potential The general idea of type-based amortized analysis for functional programs has been introduced in [1] as follows. First, inductive data structures are statically annotated with a positive rational number q to define a non-negative potential $\Phi(n) = q \cdot n$ as a function of the size n of the data. Second, the potential is shown to be sufficient to pay for all operations that are performed on this data structure during any possible evaluation of the program. The initial potential (summed over all input data) then describes an upper bound on the resource costs. We illustrate the idea by analyzing the heap-space consumption of the function *attach* below.

```
attach (x,l) = match l with | nil → nil | (y::ys) → (x,y)::( attach (x,ys))
```

It takes an integer and a list of integers and returns a list of pairs of integers in which the first argument is paired with each element of the list. If we assume that a list element for a pair of integers has size 3 (two cells for the integers, one for the pointer to the next element) then the heap-space cost of an evaluation of *attach(x,l)* is $3|l|$ memory cells.

In order to infer an upper bound on the heap-space usage of the function we annotate the type of *attach* with a priori unknown resource-annotations s, s', q and p that range over non-negative rational numbers. The intuitive meaning of the resulting type *attach:* $(\mathit{int}, L^q(\mathit{int})) \xrightarrow{s/s'} L^p(\mathit{int}, \mathit{int})$ is as follows: to evaluate *attach(x,l)* one needs q memory cells per element in the list *l* and s additional memory cells. After the evaluation there are s' memory cells and p cells per element of the returned list left. We say that the list *l* has potential $\Phi(l, q) = q \cdot |l|$ and that *l' = attach(x,l)* has potential $\Phi(l', p) = p \cdot |l'|$.

The problem of computing a resource bound then amounts to finding valid instantiations of the resource variables, i.e., a potential that suffices to cover the costs of any possible evaluation. The validity of an instantiation can be verified statically in a sound albeit not complete type-based analysis of the program text. A valid resource annotation for *attach* can be obtained by setting $q = 3$ and $s = s' = p = 0$. The computed upper bound on the heap-space costs is then $3n$ where n is length of the input list. Another possible instantiation would be $q = 6$, $p = 3$, and $s = s' = 0$. The resulting typing of *attach* could be used for the inner occurrence of *attach* to type an expression like *attach(x,attach(z,ys))*. The associated upper bound on the heap-space costs for the evaluation of the expression is then $6|ys|$.

The use of linear potential functions relieves one of the burden of having to manipulate symbolic expressions during the analysis by a priori fixing their format. This gives rise to a particularly efficient inference algorithm for the type annotations. It works like a standard constraint-based type inference in which simple linear constraints are collected as each type rule is applied. The constraints are then solved by linear programming. To see the basic idea, consider the function *attach* in which expressions of type list are annotated with variables $q, p, r, \ldots$ that range over $\mathbb{Q}^+$. The intended meaning of l^q is that l is of type $L^q(A)$ for some type A.

attach (x, l^q) = **match** $\mathrm{l}^{q'}$ **with** | nil $\rightarrow$ nil^p | (y :: ys^r) $\rightarrow$ ((x,y)::(attach (x, ys^q))^p)^p

The syntax-directed inference then computes inequalities like $q' + s \geq 3 + p + s$. It expresses the fact that the potential q' of the first list element and the initial potential s need to cover the costs for the cons operation (3 memory cells), the potential p of a list element of the result, and the input potential s of the recursive call. To pay the cost during the recursion we require the annotation of the function arguments and the result of the recursive call to match their specification (q and p in the case of *attach*). The function is then used *resource-monomorphically*.

Polynomial Potential Our previous work [9] showed that an automatic amortized analysis can also be used to derive *polynomial* resource bounds by extracting *linear* inequalities from a program. The main innovation is the use of potential-functions of the form $\sum_{i=1,\ldots,k} q_i \binom{n}{i}$ with $q_i \geq 0$. They are attached to inductive data structures via type annotations of the form $\vec{q} = (q_1, \ldots, q_k)$ with $q_i \in \mathbb{Q}^+$. For instance, the typing $l{:}L^{(3,2,1)}(\mathit{int})$, defines the potential $\Phi(l, (3,2,1)) = 3|l| + 2\binom{|l|}{2} + 1\binom{|l|}{3}$.

The use of binomial coefficients rather than powers of variables has many advantages as discussed in [9]. In particular, the identity $q_1 + \sum_{i=1,\ldots,k-1} q_{i+1}\binom{n}{i} + \sum_{i=1,\ldots,k} q_i\binom{n}{i} = \sum_{i=1,\ldots,k} q_i\binom{n+1}{i}$ gives rise to a local typing rule for *cons match* which naturally allows the typing of both recursive calls and other calls to subordinate functions in branches of a pattern match. This identity forms the mathematical basis of the *additive shift* $\lhd$ of a type annotation which is defined by $\lhd(q_1, \ldots, q_k) = (q_1 + q_2, \ldots, q_{k-1} + q_k, q_k)$. It appears, e.g., in the typing *tail*:$L^{\vec{q}}(\mathit{int}) \xrightarrow{0/q_1} L^{\lhd(\vec{q})}(\mathit{int})$ of the function *tail* that removes the first element from a list. The idea underlying the additive shift is that the potential resulting from the contraction *xs*:$L^{\lhd(\vec{q})}(\mathit{int})$ of a list *(x::xs)*:$L^{\vec{q}}(\mathit{int})$ (usually in a pattern match) is used for three purposes: (i) to pay the constant cost after and before the recursive calls (q_1), (ii) to fund calls to auxiliary functions ($(q_2, \ldots, q_n)$), and (iii) to pay for the recursive calls ($(q_1, \ldots, q_n)$).

To see how the polynomial potential annotations are used to compute polynomial resource bounds, consider the function *pairs* that computes the two-element subsets of a given set (representing sets as tuples or lists).

pairs l = **match** l **with** | nil $\rightarrow$ nil | (x :: xs) $\rightarrow$ append(attach(x,xs), pairs xs)

The function *append* consumes 3 memory cells for every element in the first argument. Similar to *attach* we can compute a tight resource bound for *append* by inferring the type *append:* $(L^{(3)}(\mathit{int},\mathit{int}), L^{(0)}(\mathit{int},\mathit{int})) \xrightarrow{0/0} L^{(0)}(\mathit{int},\mathit{int})$.

The evaluation of the expression *pairs(l)* consumes 6 memory cells per element of every sub-list (suffix) of l. The inferred type for *pairs* is $L^{(0,6)}(\mathit{int}) \xrightarrow{0/0} L^{(0)}(\mathit{int},\mathit{int})$.

It states that a list l in an expression *pairs(l)* has the potential $\Phi(l,(0,6)) = 0 \cdot |l| + 6 \cdot \binom{|l|}{2}$ and thus furnishes a tight upper bound on the heap-space usage. To type the function's body, the additive shift assigns the type $xs{:}L^{(0+6,6)}(\mathit{int})$ to the variable *xs* in the pattern match. The potential is shared between the two occurrences of *xs* in the following expression by using $xs{:}L^{(6,0)}(\mathit{int})$ to pay for *append* and *attach* (ii) and using $xs{:}L^{(0,6)}(\mathit{int})$ to pay for the recursive call of *pairs* (iii); the constant costs (i) are zero.

To compute the bound, we start with an annotation with resource variables as before.

$$\begin{array}{l} \text{pairs l} = \textbf{match } \text{l}^{(q_1,q_2)} \textbf{ with } \mid \text{nil} \to \text{nil} \\ \qquad\qquad \mid (\text{x} :: \text{xs}^{(p_1,p_2)}) \to \text{append(attach(x,xs}^{(r_1,r_2)}\text{),pairs xs}^{(s_1,s_2)}) \end{array}$$

The constraints that our type system computes include $q_2{\geq}p_2$ and $q_1{+}q_2{\geq}p_1$ (additive shift); $p_1{=}r_1{+}s_1$ and $p_2{=}r_2{+}s_2$ (sharing between two variables); $r_1{\geq}6$ (pay for non-recursive function calls); $q_1{=}s_1$, $q_2{=}s_2$ (pay for the recursive call). This system is solvable by $q_2 = s_2 = p_1 = p_2 = r_1 = 6$ and $q_1 = s_1 = r_2 = 0$.

Polymorphic Recursion As in the linear case, we require in the constraint system that the type of the recursive call of *pairs* matches its specification ($q_i = s_i$). But other than in the linear case, such a resource-monomorphic approach results in an unsolvable linear program for many non-tail-recursive functions with a super linear resource behavior. We illustrate this with the function *pairs'* that is a modification of *pairs* in which we permute the arguments of *append* and hence replace the expression in the cons-branch of the pattern match with *append(pairs' xs,attach(x,xs))*. The heap-space usage of *pairs'* is $3\binom{n}{2} + 3\binom{n}{3}$ since *append* is called with the intermediate results of *pairs'* in the first argument and thus consumes $\sum_{2\leq i<n}\binom{i}{2} = \binom{n}{3}$ memory cells.

The resource-polymorphic system determines an exact heap-space bound for the function *pairs'* by computing the typing $L^{(0,3,3)}(\mathit{int}) \xrightarrow{0/0} L^{(0)}(\mathit{int},\mathit{int})$. Similar to the case of *pairs* the additive shift assigns the type $L^{(3,6,3)}(\mathit{int})$ to *xs* in the cons-branch. The linear potential *xs:*$L^{(3,0,0)}(\mathit{int})$ is passed on to the occurrence of *xs* in *attach*. But in order to pay the costs of *append* we have to assign a linear potential to the result of the recursive call and thus use the alternate typing *pairs':* $L^{(0,6,3)}(\mathit{int}) \xrightarrow{0/0} L^{(3)}(\mathit{int},\mathit{int})$. The need of passing on potential of degree at most $k-1$ to the output of a function with a resource consumption of degree k is quite common in typical functions. It is present in the derivation of time bounds for most non-tail-recursive functions that we considered, e.g., quick sort and insertion sort. The classic (resource-monomorphic) inference approach of requiring the type of the recursive call to match its specification fails for these functions and it was a non-trivial problem to address it with an efficient solution.

Cost-Free Resource Metric Our pragmatic approach is to introduce a special *cost-free* resource metric that assigns zero costs to every evaluation step. A cost-free function type *f:* $A \xrightarrow{a/a'} B$ then describes how to pass potential from x to $f(x)$ without paying for resource usage. Any concrete typing for a given resource metric can be superposed with a *cost-free* typing to obtain another typing for the given resource metric (cf. solutions of inhomogeneous systems by superposition with homogeneous solutions in lin. algebra).

We illustrate the idea using *pairs'*. For $A{=}(\mathit{int},\mathit{int})$, we derive the cost-free types *attach*: $(\mathit{int}, L^{(3)}(\mathit{int})) \xrightarrow{0/0} L^{(3)}(A)$ and *append*: $(L^{(3)}(A), L^{(3)}(A)) \xrightarrow{0/0} L^{(3)}(A)$. The type inference for, e.g., *attach* works as outlined above with the inequality $q' + s \geq 3 + p + s$ replaced with $q' + s \geq p + s$. Similar, we can assign *pairs'* the cost-free type $L^{(0,3)}(\mathit{int}) \xrightarrow{0/0} L^{(3)}(\mathit{int}, \mathit{int})$. The typing $\mathit{xs}{:}L^{(3,3)}(\mathit{int})$ that results from the additive shift is used as $\mathit{xs}{:}L^{(3,0)}(\mathit{int})$ in *attach* and as $\mathit{xs}{:}L^{(0,3)}(\mathit{int})$ in the recursive call.

If we now want to infer the type of a function with respect to some cost metric then we deal with recursive calls by requiring them to match the functions type specification and to optionally pass potential to the result via a cost-free type. The cost-free type is then inferred resource-monomorphically. In the case of the heap-space consumption of *pairs'* we would first infer that the recursive call has to be of the form $L^{(0+q_1,3+q_2,3)}(\mathit{int}) \to L^{(0+p_1)}(\mathit{int}, \mathit{int})$ such that $L^{(q_1,q_2)}(\mathit{int}) \to L^{(p_1)}(\mathit{int}, \mathit{int})$ is a cost-free type. We then infer like in the linear case that $q_1 = 0$ and $q_2 = p_1 = 3$.

This method cannot infer every resource-polymorphic typing with respect to declarative type derivations with polymorphic recursion. This would mean to start with a (possibly infinite) set of annotated types for each function and to justify each function type with a type derivation that uses types from the initial set. With respect to this declarative view, the inference algorithm in this paper can compute every set of types for a function f that has the form $\Sigma(f) = \{T + q \cdot T_i \mid q \in \mathbb{Q}^+, 1 \leq i \leq m\}$ for a resource-annotated function type T, cost-free function types T_i, and m recursive calls of f in its function body. Since many resource-polymorphic type derivations feature a set of function types of this format, our approach leads to an effective inference method. In the algorithmic type rules (Fig. 3) we directly integrated the above format of $\Sigma(f)$ in the rule T:FunApp for function applications to enable an efficient inference.

3 Resource Aware ML

RAML (Resource Aware ML) is a first-order functional language with ML-style syntax, booleans, integers, pairs, lists, recursion and pattern match.

To simplify typing rules in this paper, we define the following *expressions of RAML* to be in *let normal form*. In the implementation we allow unrestricted expressions. One can use every binary operation *binop* whose worst-case cost is bounded by a constant.

$$\begin{aligned} e ::= {} & () \mid \mathit{True} \mid \mathit{False} \mid n \mid x \mid x_1\ \mathit{binop}\ x_2 \mid f(x_1, \ldots, x_n) \mid \mathit{let}\ x = e_1\ \mathit{in}\ e_2 \\ & \mid \mathit{if}\ x\ \mathit{then}\ e_t\ \mathit{else}\ e_f \mid (x_1, x_2) \mid \mathit{match}\ x\ \mathit{with}\ (x_1, x_2) \to e \\ & \mid \mathit{nil} \mid \mathit{cons}(x_h, x_t) \mid \mathit{match}\ x\ \mathit{with}\ \mid \mathit{nil} \to e_1 \mid \mathit{cons}(x_h, x_t) \to e_2 \end{aligned}$$

In the implementation of RAML we included a destructive pattern match and the extended version of [9] describes how polynomial potential can be applied to tree-like data types. The inference algorithm can easily be adopted to handle these extensions.

We define the well-typed expressions of RAML by assigning a *simple type*, a usual ML type without resource annotations, to well-typed expressions. Simple types are data types and first-order types as given by the grammars below.

$$A ::= \mathit{unit} \mid \mathit{bool} \mid \mathit{int} \mid L(A) \mid (A, A) \qquad\qquad F ::= A \to A$$

A *typing context* Γ is a partial, finite mapping from variable identifiers to data types. A *signature* Σ is a finite, partial mapping of function identifiers to first-order types. The typing judgment $\Gamma \vdash_\Sigma e : A$ states that the expression e has type A under the signature Σ in the context Γ. The typing rules that define the typing judgment are standard and identical with the resource-annotated typing rules from §5 if the resource annotations are omitted. A *RAML program* consists of a signature Σ and a family $(e_f, y_f)_{f \in \mathrm{dom}(\Sigma)}$ of expressions with a variable identifier such that $y_f{:}A \vdash_\Sigma e_f{:}B$ if $\Sigma(f) = A \to B$.

4 Operational Semantics

We define a big-step operational semantics that measures the quantitative resource consumption of programs. It is parametric in the resource of interest and can measure every quantity whose usage in a single evaluation step can be bounded by a constant. The actual constants for a step on a specific system architecture can be derived by analyzing the translation of the step in the compiler implementation for that architecture [5].

The semantics is formulated with respect to a stack and a heap: A value $v \in \mathrm{Val}$ is either a location $l \in \mathrm{Loc}$, a boolean constant b, an integer n, a null value NULL or a pair of values (v_1, v_2). A *heap* is a finite partial mapping $\mathcal{H} : \mathrm{Loc} \to \mathrm{Val}$ from locations to values. A *stack* is a finite partial mapping $\mathcal{V} : \mathrm{VID} \to \mathrm{Val}$ from variable identifiers to values. Since we also consider resources like memory that can become available during an evaluation, we have to track the *watermark* of the resource usage, i.e., the maximal number of resources units that are simultaneously used during an evaluation. In order to derive a watermark of a sequence of evaluations from the watermarks of the sub evaluations one has also to take into account the number of resource units that are available after each sub evaluation.

The operational evaluation rules in Fig. 1 thus define an evaluation judgment of the form $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow v, \mathcal{H}' \mid (q, q')$ expressing the following. If the stack $\mathcal{V}$ and the initial heap $\mathcal{H}$ are given then the expression e evaluates to the value v and the new heap $\mathcal{H}'$. In order to evaluate e one needs at least $q \in \mathbb{Q}^+$ resource units and after the evaluation there are at least $q' \in \mathbb{Q}^+$ resource units available. The actual resource consumption is then $\delta = q - q'$. The quantity δ is negative if resources become available.

In contrast to similar versions in earlier works there is at most one pair (q, q') such that $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow v, \mathcal{H}' \mid (q, q')$ for an expression e and fixed $\mathcal{H}$ and $\mathcal{V}$. The non-negative number q is the watermark of simultaneous resources usage during the evaluation.

It is handy to view the pairs (q, q') in the evaluation judgments as elements of a monoid[2] $\mathcal{R} = (\mathbb{Q}^+ \times \mathbb{Q}^+, \cdot)$. The neutral element is $(0, 0)$ which means that resources are neither used nor restituted. The operation $(q, q') \cdot (p, p')$ defines how to account for an evaluation consisting of evaluations whose resource consumptions are defined by (q, q') and (p, p'), respectively. We define

$$(q, q') \cdot (p, p') = \begin{cases} (q + p - q',\ p') \text{ if } q' \le p \\ (q,\ p' + q' - p) \text{ if } q' > p \end{cases}$$

The intuition is that we need q resource units to perform the first evaluation after which q' restituted units remain. The second operation needs then p units. If $q' \le p$ then we additionally need $p - q'$ resources to pay for both evaluations and have p' resources left

[2] It is possible to define the evaluation more abstractly with respect to an arbitrary monoid M.

$$\frac{}{\mathcal{V},\mathcal{H} \vdash () \rightsquigarrow \textsc{Null},\mathcal{H} \mid K^{\text{unit}}}\ \textsc{E:ConstU} \qquad \frac{x_1, x_2 \in \text{dom}(\mathcal{V}) \qquad v = op(\mathcal{V}(x_1),\mathcal{V}(x_2))}{\mathcal{V},\mathcal{H} \vdash x_1\ op\ x_2 \rightsquigarrow v,\mathcal{H} \mid K^{\text{op}}}\ \textsc{E:BinOp}$$

$$\frac{n \in \mathbb{Z}}{\mathcal{V},\mathcal{H} \vdash n \rightsquigarrow n,\mathcal{H} \mid K^{\text{int}}}\ \textsc{E:ConstI} \qquad \frac{\mathcal{V}(x)=v \qquad [y_f \mapsto v],\mathcal{H} \vdash e_f \rightsquigarrow v',\mathcal{H}' \mid (q,q')}{\mathcal{V},\mathcal{H} \vdash f(x) \rightsquigarrow v',\mathcal{H}' \mid K_1^{\text{app}} \cdot (q,q') \cdot K_2^{\text{app}}}\ \textsc{E:FunApp}$$

$$\frac{b \in \{\mathit{True}, \mathit{False}\}}{\mathcal{V},\mathcal{H} \vdash b \rightsquigarrow b,\mathcal{H} \mid K^{\text{bool}}}\ \textsc{E:ConstB} \qquad \frac{\mathcal{V}(x) = \mathit{True} \qquad \mathcal{V},\mathcal{H} \vdash e_t \rightsquigarrow v,\mathcal{H}' \mid (q,q')}{\mathcal{V},\mathcal{H} \vdash \mathit{if}\ x\ \mathit{then}\ e_t\ \mathit{else}\ e_f \rightsquigarrow v,\mathcal{H}' \mid K_1^{\text{conT}} \cdot (q,q') \cdot K_2^{\text{conT}}}\ \textsc{E:CondT}$$

$$\frac{\mathcal{V}(x) = \mathit{False} \qquad \mathcal{V},\mathcal{H} \vdash e_f \rightsquigarrow v,\mathcal{H}' \mid (q,q')}{\mathcal{V},\mathcal{H} \vdash \mathit{if}\ x\ \mathit{then}\ e_t\ \mathit{else}\ e_f \rightsquigarrow v,\mathcal{H}' \mid K_1^{\text{conF}} \cdot (q,q') \cdot K_2^{\text{conF}}}\ \textsc{E:CondF}$$

$$\frac{\mathcal{V},\mathcal{H} \vdash e_1 \rightsquigarrow v_1,\mathcal{H}_1 \mid (q,q') \qquad \mathcal{V}[x \mapsto v_1],\mathcal{H}_1 \vdash e_2 \rightsquigarrow v_2,\mathcal{H}_2 \mid (p,p')}{\mathcal{V},\mathcal{H} \vdash \mathit{let}\ x = e_1\ \mathit{in}\ e_2 \rightsquigarrow v_2,\mathcal{H}_2 \mid K_1^{\text{let}} \cdot (q,q') \cdot K_2^{\text{let}} \cdot (p,p') \cdot K_3^{\text{let}}}\ \textsc{E:Let}$$

$$\frac{x_1, x_2 \in \text{dom}(\mathcal{V}) \qquad v = (\mathcal{V}(x_1),\mathcal{V}(x_2))}{\mathcal{V},\mathcal{H} \vdash (x_1,x_2) \rightsquigarrow v,\mathcal{H} \mid K^{\text{pair}}}\ \textsc{E:Pair} \qquad \frac{}{\mathcal{V},\mathcal{H} \vdash \mathit{nil} \rightsquigarrow \textsc{Null},\mathcal{H} \mid K^{\text{nil}}}\ \textsc{E:Nil}$$

$$\frac{x \in \text{dom}(\mathcal{V})}{\mathcal{V},\mathcal{H} \vdash x \rightsquigarrow \mathcal{V}(x),\mathcal{H} \mid K^{\text{var}}}\ \textsc{E:Var} \qquad \frac{x_h, x_t \in \text{dom}(\mathcal{V}) \qquad v = (\mathcal{V}(x_h),\mathcal{V}(x_t)) \qquad l \notin \text{dom}(\mathcal{H})}{\mathcal{V},\mathcal{H} \vdash \mathit{cons}(x_h,x_t) \rightsquigarrow l,\mathcal{H}[l \mapsto v] \mid K^{\text{cons}}}\ \textsc{E:Cons}$$

$$\frac{\mathcal{V}(x) = (v_1,v_2) \qquad \mathcal{V}[x_1 \mapsto v_1, x_2 \mapsto v_2],\mathcal{H} \vdash e \rightsquigarrow v,\mathcal{H}' \mid (q,q')}{\mathcal{V},\mathcal{H} \vdash \mathit{match}\ x\ \mathit{with}\ (x_1,x_2) \to e \rightsquigarrow v,\mathcal{H}' \mid K_1^{\text{matP}} \cdot (q,q') \cdot K_2^{\text{matP}}}\ \textsc{E:MatP}$$

$$\frac{\mathcal{V}(x) = \textsc{Null} \qquad \mathcal{V},\mathcal{H} \vdash e_1 \rightsquigarrow v,\mathcal{H}' \mid (q,q')}{\mathcal{V},\mathcal{H} \vdash \mathit{match}\ x\ \mathit{with}\ \mid \mathit{nil} \to e_1 \mid \mathit{cons}(x_h,x_t) \to e_2 \rightsquigarrow v,\mathcal{H}' \mid K_1^{\text{matN}} \cdot (q,q') \cdot K_2^{\text{matN}}}\ \textsc{E:MatN}$$

$$\frac{\mathcal{V}(x)=l \qquad \mathcal{H}(l)=(v_h,v_t) \qquad \mathcal{V}[x_h \mapsto v_h, x_t \mapsto v_t],\mathcal{H} \vdash e_2 \rightsquigarrow v,\mathcal{H}' \mid (q,q')}{\mathcal{V},\mathcal{H} \vdash \mathit{match}\ x\ \mathit{with}\ \mid \mathit{nil} \to e_1 \mid \mathit{cons}(x_h,x_t) \to e_2 \rightsquigarrow v,\mathcal{H}' \mid K_1^{\text{matC}} \cdot (q,q') \cdot K_2^{\text{matC}}}\ \textsc{E:MatC}$$

Fig. 1. Big-step operational semantics.

in the end. If $q' > p$ then q units suffices to perform both evaluations. Additionally, the $q' - p$ units that are not needed for the second evaluation are added to the resources becoming finally available. If resources are never restituted (as with time) then we can restrict to elements of the form $(q,0)$ and $(q,0) \cdot (p,0)$ is just $(q+p,0)$.

We identify (positive and negative) rational numbers with elements of $\mathcal{R}$ as follows: $q \geq 0$ denotes $(q,0)$ and $q < 0$ denotes $(0,-q)$. This notation avoids case distinctions in the evaluation rules since the constants K that appear in the rules might be negative.

Partial Evaluations A shortcoming of classic big-step operational semantics is that it does not provide judgments for evaluations that diverge. This is problematic if one intends to prove statements for divergent and convergent computations.

A straightforward remedy is to use a small-step semantics. But in the context of resource analysis, the use of big-step rules seems to be more favorable. First, big-step rules can more directly axiomatize the resource behavior of compiled code on specific machines. Secondly, it allows for shorter and less syntactic proofs.

An alternative approach is to use coinductively defined big-step semantics [10, 11].

$$\frac{}{\mathcal{V},\mathcal{H} \vdash e \rightsquigarrow \mid 0}\ \textsc{P:Zero} \qquad \frac{b \in \{\mathit{True}, \mathit{False}\}}{\mathcal{V},\mathcal{H} \vdash b \rightsquigarrow \mid K^{\text{bool}}}\ \textsc{P:ConstB} \qquad \frac{}{\mathcal{V},\mathcal{H} \vdash () \rightsquigarrow \mid K^{\text{unit}}}\ \textsc{P:ConstU}$$

$$\frac{n \in \mathbb{Z}}{\mathcal{V},\mathcal{H} \vdash n \rightsquigarrow \mid K^{\text{int}}}\ \textsc{P:ConstI} \qquad \frac{x \in \text{dom}(\mathcal{V})}{\mathcal{V},\mathcal{H} \vdash x \rightsquigarrow \mid K^{\text{var}}}\ \textsc{P:Var} \qquad \frac{x_1, x_2 \in \text{dom}(\mathcal{V})}{\mathcal{V},\mathcal{H} \vdash (x_1,x_2) \rightsquigarrow \mid K^{\text{pair}}}\ \textsc{P:Pair}$$

$$\frac{\mathcal{V}(x) = v \qquad [y_f \mapsto v],\mathcal{H} \vdash e_f \rightsquigarrow \mid q}{\mathcal{V},\mathcal{H} \vdash f(x) \rightsquigarrow \mid K_1^{\text{app}} + q}\ \textsc{P:FunApp} \qquad \frac{\mathcal{V},\mathcal{H} \vdash e_1 \rightsquigarrow \mid q}{\mathcal{V},\mathcal{H} \vdash \mathit{let}\, x = e_1\ \mathit{in}\, e_2 \rightsquigarrow \mid K_1^{\text{let}}{+}q}\ \textsc{P:Let1}$$

$$\frac{\mathcal{V},\mathcal{H} \vdash e_1 \rightsquigarrow v_1, \mathcal{H}_1 \mid (q,q') \qquad \mathcal{V}[x{\mapsto}v_1],\mathcal{H}_1 \vdash e_2 \rightsquigarrow \mid p \qquad K_1^{\text{let}}{\cdot}(q,q'){\cdot}K_2^{\text{let}}{\cdot}(p,0){=}(r,r')}{\mathcal{V},\mathcal{H} \vdash \mathit{let}\, x = e_1\ \mathit{in}\, e_2 \rightsquigarrow \mid r}\ \textsc{P:Let2}$$

$$\frac{\mathcal{V}(x) = \mathit{True} \qquad \mathcal{V},\mathcal{H} \vdash e_t \rightsquigarrow \mid q}{\mathcal{V},\mathcal{H} \vdash \mathit{if}\, x\ \mathit{then}\, e_t\ \mathit{else}\, e_f \rightsquigarrow \mid K_1^{\text{conT}} + q}\ \textsc{P:CondT} \qquad \frac{x_1, x_2 \in \text{dom}(\mathcal{V})}{\mathcal{V},\mathcal{H} \vdash x_1\ \mathit{op}\ x_2 \rightsquigarrow \mid K^{\text{op}}}\ \textsc{P:BinOp}$$

$$\frac{\mathcal{V}(x) = \mathit{False} \qquad \mathcal{V},\mathcal{H} \vdash e_f \rightsquigarrow \mid q}{\mathcal{V},\mathcal{H} \vdash \mathit{if}\, x\ \mathit{then}\, e_t\ \mathit{else}\, e_f \rightsquigarrow \mid K_1^{\text{conF}}{+}q}\ \textsc{P:CondF} \qquad \frac{x_h, x_t \in \text{dom}(\mathcal{V})}{\mathcal{V},\mathcal{H} \vdash \mathit{cons}(x_h,x_t) \rightsquigarrow \mid K^{\text{cons}}}\ \textsc{P:Cons}$$

$$\frac{\mathcal{V}(x) = (v_1,v_2) \qquad \mathcal{V}[x_1 \mapsto v_1, x_2 \mapsto v_2],\mathcal{H} \vdash e \rightsquigarrow \mid q}{\mathcal{V},\mathcal{H} \vdash \mathit{match}\ x\ \mathit{with}\ (x_1,x_2) \to e \rightsquigarrow \mid K_1^{\text{matP}}{+}q}\ \textsc{P:MatP} \qquad \frac{}{\mathcal{V},\mathcal{H} \vdash \mathit{nil} \rightsquigarrow \mid K^{\text{nil}}}\ \textsc{P:Nil}$$

$$\frac{\mathcal{V}(x) = \textsc{Null} \qquad \mathcal{V},\mathcal{H} \vdash e_1 \rightsquigarrow \mid q}{\mathcal{V},\mathcal{H} \vdash \mathit{match}\ x\ \mathit{with}\ \mid \mathit{nil} \to e_1 \mid \mathit{cons}(x_h,x_t) \to e_2 \rightsquigarrow \mid K_1^{\text{matN}} + q}\ \textsc{P:MatN}$$

$$\frac{\mathcal{V}(x) = l \qquad \mathcal{H}(l) = (v_h,v_t) \qquad \mathcal{V}[x_h \mapsto v_h, x_t \mapsto v_t],\mathcal{H} \vdash e_2 \rightsquigarrow \mid q}{\mathcal{V},\mathcal{H} \vdash \mathit{match}\ x\ \mathit{with}\ \mid \mathit{nil} \to e_1 \mid \mathit{cons}(x_h,x_t) \to e_2 \rightsquigarrow \mid K_1^{\text{matC}} + q}\ \textsc{P:MatC}$$

Fig. 2. Partial big-step operational semantics.

However, coinductive semantics lends itself less well to formulating and proving semantic soundness theorems of the form “if the program is well-typed and the operational semantics says X then Y holds” (like Thm. 4). For example, in Leroy’s Lemmas 17-22 [11] the coinductive definition appears in the conclusion rather than as a premise.

That is why we use a novel approach to the problem here by defining a *big-step semantics for partial evaluations* that directly corresponds to the rules of the big-step semantics in Fig. 1. It defines a statement of the form $\mathcal{V},\mathcal{H} \vdash e \rightsquigarrow \mid q$ for a stack $\mathcal{V}$, a heap $\mathcal{H}$, $q \in \mathbb{Q}^+$ and an expression e. The meaning is that there is a partial evaluation of e with the stack $\mathcal{V}$ and the heap $\mathcal{H}$ that consumes q resources. Here, q is the watermark of the resource usage. We do not have to keep track of the restituted resources.

Note that the rule P:ZERO is essential for the partiality of the semantics. It can be applied at any point to stop the evaluation and thus yields to a non-deterministic evaluation judgment.

Since there might be negative constants K, the partial evaluation rules in Fig. 2 have conclusions of the form $\mathcal{V},\mathcal{H} \vdash e \rightsquigarrow \mid \max(q,0)$ to ensure non-negative values. We simply write $\mathcal{V},\mathcal{H} \vdash e \rightsquigarrow \mid q$ instead of $\mathcal{V},\mathcal{H} \vdash e \rightsquigarrow \mid \max(q,0)$ in each conclusion.

We prove that if an expression converges in a given environment then the resource-usage watermark of the evaluation is an upper bound for the resource usage of every partial evaluation of the expression in that environment.

Theorem 1. *If $\mathcal{V},\mathcal{H} \vdash e \rightsquigarrow v,\mathcal{H}' \mid (q,q')$ and $\mathcal{V},\mathcal{H} \vdash e \rightsquigarrow \mid p$ then $p \le q$.*

A stack $\mathcal{V}$ and a heap $\mathcal{H}$ are *well-formed* with respect to a context Γ if, for every $x \in \mathrm{dom}(\Gamma)$, $\mathcal{V}(x)$ is a value matching the type $\Gamma(x)$ or a location in $\mathcal{H}$ that contains a value matching $\Gamma(x)$. We then write $\mathcal{H} \vDash \mathcal{V}{:}\Gamma$. Similarly, we write $\mathcal{H} \vDash v{:}A$ if v is a value matching type A in $\mathcal{H}$. A formal definition is given in [7].

Thm. 2 states that, in a well-formed environment, every well-typed expression either diverges or evaluates to a value of the stated type. To this end we instantiate the resource constants in the rules to count the number of evaluation steps.

Theorem 2. *Let the resource constants be instantiated by $K^x = 1$, $K^x_1 = 1$ and $K^x_m = 0$ for all x and all $m > 1$. If $\Gamma \vdash_\Sigma e{:}A$ and $\mathcal{H} \vDash \mathcal{V}{:}\Gamma$ then $\mathcal{V},\mathcal{H} \vdash e \rightsquigarrow v,\mathcal{H}' \mid (n,0)$ for an $n \in \mathbb{N}$ or $\mathcal{V},\mathcal{H} \vdash e \rightsquigarrow \mid m$ for every $m \in \mathbb{N}$.*

Cost-Free Metric The type inference algorithm makes use of the *cost-free* resource metric. This is the metric in which all constants K that appear in the rules are instantiated to zero. We will use it in §5 to define a resource-polymorphic recursion where we use cost-free function types to pass potential from the argument to the result.

With the cost-free resource metric the resource usage of evaluations is always zero: If $\mathcal{V},\mathcal{H} \vdash e \rightsquigarrow v,\mathcal{H}' \mid (q,q')$ then $q = q' = 0$ and if $\mathcal{V},\mathcal{H} \vdash e \rightsquigarrow \mid q$ then $q = 0$.

5 Resource Annotated Types

Resource-annotated types are simple types where lists are annotated with non-negative vectors $\vec{p} \in \mathbb{Q}^n$. Here we only give a short definition of the potential functions defined by annotated types. More explanations can be found in [9].

Let $\vec{p} = (p_1,\ldots,p_k)$ be an annotation.The *additive shift* of $\vec{p}$ is $\lhd(\vec{p}) = (p_1 + p_2, p_2 + p_3, \ldots, p_{k-1} + p_k, p_k)$. Let $\mathcal{H}$ be a heap, A be a resource-annotated type and let v be a value matching type A in $\mathcal{H}$. The *potential* $\Phi_\mathcal{H}(v{:}A)$ is then defined as follows.

$$\Phi_\mathcal{H}(v{:}A) = 0 \text{ if } v = \textsc{Null} \text{ or } A \in \{\mathit{unit}, \mathit{int}, \mathit{bool}\}$$
$$\Phi_\mathcal{H}((v_1,v_2){:}(A_1,A_2)) = \Phi_\mathcal{H}(v_1{:}A_1) + \Phi_\mathcal{H}(v_2{:}A_2)$$
$$\Phi_\mathcal{H}(l{:}L^{\vec{p}}(A')) = p_1 + \Phi_\mathcal{H}(v'{:}A') + \Phi_\mathcal{H}(l'{:}\ L^{\lhd(\vec{p})}(A')) \text{ if } \mathcal{H}(l){=}(v',l')$$

If l_1 is a location that points to a list then we write $\mathcal{H}(l_1) = [v_1,\ldots,v_n]$ if $\mathcal{H}(l_i) = (v_i, l_{i+1})$ for $i = 1,\ldots,n$ and $l_{n+1} = \textsc{Null}$. If $l_1 = \textsc{Null}$ then $\mathcal{H}(l_1) = []$. Thm. 3 shows how to express the potential $\Phi_\mathcal{H}(v{:}A)$ of a value v with respect the heap $\mathcal{H}$ and a matching annotated type A in terms of polynomials in the lengths of the lists that are reachable from v. A proof can be found in the extended version of [9].

Theorem 3. *Let $\mathcal{H}$ be a heap and let $\mathcal{H}(l) = [v_1\ldots,v_n]$ be a list of length n. Then $\Phi_\mathcal{H}(l{:}L^{\vec{p}}(A)) = \sum_{i=1}^{k} p_i \binom{n}{i} + \sum_{i=1}^{n} \Phi_\mathcal{H}(v_i{:}A)$.*

As in the case of simple types, a *typing context* is a finite partial mapping from variable identifiers to annotated data types. The potential of a context Γ with respect to a heap $\mathcal{H}$ and a stack $\mathcal{V}$ is $\Phi_{\mathcal{V},\mathcal{H}}(\Gamma) = \sum_{x \in \mathrm{dom}(\Gamma)} \Phi_\mathcal{H}(\mathcal{V}(x){:}\Gamma(x))$.

Resource-annotated first-order types have the form $A \xrightarrow{q/q'} B$ for $q, q' \in \mathbb{Q}^+$ and annotated data types A, B. A *resource-annotated signature* Σ is a finite, partial mapping from function identifiers to resource-annotated first-order types.

A *resource-annotated typing judgment* has the form $\Sigma; \Gamma \;{}^{k}\!\vdash^{q}_{q'}\; e{:}A$ where e is a RAML expression, $k \in \mathbb{N}^+$ is the length of the list annotations, $q, q' \in \mathbb{Q}^+$ are non-negative rational numbers, Σ is a resource-annotated signature, Γ is a resource-annotated context and A is a resource-annotated data type. The intended meaning of this judgment is that if there are more than $q + \Phi(\Gamma)$ resource units available then this is sufficient to evaluate e and there are more than $q' + \Phi(v{:}A)$ resource units if e evaluates to the value v.

A RAML program with resource-annotated types of degree k consists of a resource-annotated signature Σ and a family $(e_f, y_f)_{f \in \mathrm{dom}(\Sigma)}$ of expressions with variables identifiers such that for each e_f we have $\Sigma; y_f{:}A \;{}^{k}\!\vdash^{q}_{q'}\; e_f{:}B$ if $\Sigma(f) = A \xrightarrow{q/q'} B$.

We write $\Sigma; \Gamma \;{}^{cf(k)}\!\vdash^{q}_{q'}\; e{:}A$ to refer to cost-free type judgments where all constants K in the rules are zero. It is used to define a resource-polymorphic recursion where we use cost-free function types to pass potential from the argument to the result (see §2).

In the typing rules in Fig. 3 we write $e[z/x]$ to denote the expression e with all free occurrences of the variable x replaced with the variable z. We assume that a fixed but arbitrary global resource-annotated signature Σ is given. Furthermore, there is the implicit constraint $q \geq 0$ for every resource annotation q.

The rules are mostly algorithmic versions of the typing rules in [9]. The most important difference is the rule T:FunApp which enables resource-polymorphic recursion. It states that one can add any cost-free typing of the function body to the function type that is given by the signature Σ. The signature Σ_{cf} is a fresh signature such that $(e_f, y_f)_{f \in \Sigma_{cf}}$ is a valid RAML program with cost-free types of degree $k - 1$. It can differ in every application of the rule. The idea is as follows. To pay for the resource costs of a function call $f(x)$, the available potential ($\Phi(x{:}B) + q$) must meet the requirements of the functions' signature ($\Phi(x{:}B') + p$). Additionally available potential ($\Phi(x{:}B_{cf}) + p_{cf}$) can be passed to a cost-free typing of the function body. The potential after the function call ($\Phi(f(x){:}A) + q'$) is then the sum of the potentials that are assigned by the cost-free typing ($\Phi(f(x){:}A_{cf}) + p_{cf}$) and by the function signature ($\Phi(f(x){:}A') + p$). As a result, $f(x)$ can be used resource-polymorphically with a specific typing for each recursive call while the resource monomorphic function signature enables an efficient type inference.

The *sharing relation* $\curlyvee$ defines how potential can be shared between multiple occurrences of a variable. Intuitively, if $\curlyvee(A \mid A_1, A_2)$ holds then $x{:}A$ can be used twice, once with type A_1 and once with type A_2. We define $\curlyvee(A \mid A, A)$ if $A \in \{\mathit{unit}, \mathit{bool}, \mathit{int}\}$; $\curlyvee(L^{\vec{p}}(A) \mid L^{\vec{q}}(A_1), L^{\vec{r}}(A_2))$ if $\curlyvee(A \mid A_1, A_2)$ and $\vec{p} = \vec{q} + \vec{r}$; and $\curlyvee((A, B) \mid (A_1, B_1), (A_2, B_2))$ if $\curlyvee(X \mid X_1, X_2)$ for $X = A, B$. The sharing relation is analogously extended to contexts $\Gamma, \Gamma_1, \Gamma_2$ with $\mathrm{dom}(\Gamma) = \mathrm{dom}(\Gamma_1) = \mathrm{dom}(\Gamma_2)$ in a per element way.

A data type A is a *subtype* of a data type B, $A <: B$, only if A and B are structurally identical, and if $\Phi(v{:}A) \geq \Phi(v{:}B)$ holds for every value v. We define $C <: C$ if $C \in \{\mathit{unit}, \mathit{bool}, \mathit{int}\}$; $(A_1, A_2) <: (B_1, B_2)$ if $A_1 <: B_1$ and $A_2 <: B_2$; and $L^{\vec{p}}(A) <: L^{\vec{q}}(B)$ if $A <: B$ and $\vec{p} \geq \vec{q}$.

$$\frac{q \geq q' + K^{\text{var}}}{\mathit{\Gamma}, x{:}A \;{}^{k}{\vdash}^{q}_{q'}\; x : A}\;\textsc{T:Var} \qquad \frac{q \geq q' + K^{\text{unit}}}{\mathit{\Gamma} \;{}^{k}{\vdash}^{q}_{q'}\; (){:}\mathit{unit}}\;\textsc{T:ConstU} \qquad \frac{n \in \mathbb{Z} \qquad q \geq q' + K^{\text{int}}}{\mathit{\Gamma} \;{}^{k}{\vdash}^{q}_{q'}\; n : \mathit{int}}\;\textsc{T:ConstI}$$

$$\frac{b \in \{\mathit{True}, \mathit{False}\} \qquad q{\geq}q'{+}K^{\text{bool}}}{\mathit{\Gamma} \;{}^{k}{\vdash}^{q}_{q'}\; b{:}\mathit{bool}}\;\textsc{T:ConstB} \qquad \frac{op \in \{\mathit{or}, \mathit{and}\} \qquad q \geq q' + K^{\text{op}}}{\mathit{\Gamma}, x_1{:}\mathit{bool}, x_2{:}\mathit{bool} \;{}^{k}{\vdash}^{q}_{q'}\; x_1 \; op \; x_2 : \mathit{bool}}\;\textsc{T:BinOpB}$$

$$\frac{q \geq q' + K^{\text{pair}}}{\mathit{\Gamma}, x_1{:}A_1, x_2{:}A_2 \;{}^{k}{\vdash}^{q}_{q'}\; (x_1, x_2){:}(A_1, A_2)}\;\textsc{T:Pair} \qquad \frac{op \in \{+, -, *, \ldots\} \qquad q{\geq}q'{+}K^{\text{op}}}{\mathit{\Gamma}, x_1{:}\mathit{int}, x_2{:}\mathit{int} \;{}^{k}{\vdash}^{q}_{q'}\; x_1 \; op \; x_2 : \mathit{int}}\;\textsc{T:BinOpI}$$

$$\frac{k = 1 \qquad \mathit{\Sigma}(f) = B \xrightarrow{p/p'} A \qquad q = p + c + K^{\text{app}}_1 \qquad q' = p' + c - K^{\text{app}}_2}{\mathit{\Gamma}, x{:}B \;{}^{k}{\vdash}^{q}_{q'}\; f(x) : A}\;\textsc{T:FunApp1}$$

$$\frac{\begin{array}{c}\mathit{\Sigma}(f){=}B' \xrightarrow{p/p'} A' \qquad \curlyvee(A \,|\, A', A_{cf}) \qquad \curlyvee(B \,|\, B', B_{cf}) \qquad \mathit{\Sigma}_{cf}(f){=}B_{cf} \xrightarrow{p_{cf}/p'_{cf}} A_{cf} \\ q{=}p{+}p_{cf}{+}c{+}K^{\text{app}}_1 \qquad q'{=}p'{+}p'_{cf}{+}c{-}K^{\text{app}}_2 \qquad \mathit{\Sigma}_{cf}; y_f{:}B_{cf} \;{}^{cf(k-1)}{\vdash}^{p_{cf}}_{p'_{cf}}\; e_f{:}A_{cf}\end{array}}{\mathit{\Gamma}, x{:}B \;{}^{k}{\vdash}^{q}_{q'}\; f(x) : A}\;\textsc{T:FunApp}$$

$$\frac{\begin{array}{c}q \geq p_1 + K^{\text{let}}_1 \qquad p'_1 \geq p_2 + K^{\text{let}}_2 \qquad p'_2 \geq q' + K^{\text{let}}_3 \qquad \curlyvee(\mathit{\Delta} \mid \mathit{\Delta}_1, \mathit{\Delta}_2) \\ \mathrm{Var}(\mathit{\Gamma}_1) \cap \mathrm{Var}(\mathit{\Gamma}_2) = \emptyset \qquad \mathit{\Gamma}_1, \mathit{\Delta}_1 \;{}^{k}{\vdash}^{p_1}_{p'_1}\; e_1{:}B \qquad \mathit{\Gamma}_2, \mathit{\Delta}_2, x{:}B \;{}^{k}{\vdash}^{p_2}_{p'_2}\; e_2{:}A\end{array}}{\mathit{\Gamma}_1, \mathit{\Gamma}_2, \mathit{\Delta} \;{}^{k}{\vdash}^{q}_{q'}\; \mathit{let}\; x = e_1 \;\mathit{in}\; e_2 : A}\;\textsc{T:Let}$$

$$\frac{\begin{array}{c}q \geq p_t + K^{\text{conT}}_1 \qquad q \geq p_f + K^{\text{conF}}_1 \qquad p'_t \geq q' + K^{\text{conT}}_2 \\ p'_f \geq q' + K^{\text{conF}}_2 \qquad A_i <: A \text{ for } i = 1, 2 \qquad \mathit{\Gamma} \;{}^{k}{\vdash}^{p_t}_{p'_t}\; e_t : A_1 \qquad \mathit{\Gamma} \;{}^{k}{\vdash}^{p_f}_{p'_f}\; e_f : A_2\end{array}}{\mathit{\Gamma}, x{:}\mathit{bool} \;{}^{k}{\vdash}^{q}_{q'}\; \mathit{if}\; x \;\mathit{then}\; e_t \;\mathit{else}\; e_f : A}\;\textsc{T:Cond}$$

$$\frac{q{\geq}p{+}K^{\text{matP}}_1 \qquad p'{\geq}q'{+}K^{\text{matP}}_2 \qquad \mathit{\Gamma}, x_1{:}B_1, x_2{:}B_2 \;{}^{k}{\vdash}^{p}_{p'}\; e{:}A}{\mathit{\Gamma}, x{:}(B_1, B_2) \;{}^{k}{\vdash}^{q}_{q'}\; \mathit{match}\; x \;\mathit{with}\; (x_1, x_2) \to e : A}\;\textsc{T:MatP} \qquad \frac{q \geq q' + K^{\text{nil}}}{\mathit{\Gamma} \;{}^{k}{\vdash}^{q}_{q'}\; \mathit{nil}{:}L(A)}\;\textsc{T:Nil}$$

$$\frac{\vec{p} = (p_1, \ldots, p_k) \qquad \vec{r} \geq \lhd(\vec{p}) \qquad q \geq q' + p_1 + K^{\text{cons}} \qquad A_i <: A \text{ for } i = 1, 2}{\mathit{\Gamma}, x_h{:}A_1, x_t{:}L^{\vec{r}}(A_2) \;{}^{k}{\vdash}^{q}_{q'}\; \mathit{cons}(x_h, x_t){:}L^{\vec{p}}(A)}\;\textsc{T:Cons}$$

$$\frac{\begin{array}{c}q{+}p_1{\geq}s_c{+}K^{\text{matC}}_1 \quad q{\geq}s_n{+}K^{\text{matN}}_1 \quad s'_c{\geq}q'{+}K^{\text{matC}}_2 \quad s'_n{\geq}q'{+}K^{\text{matN}}_2 \qquad \mathit{\Gamma} \;{}^{k}{\vdash}^{s_n}_{s'_n}\; e_1{:}A_1 \\ \vec{p}{=}(p_1, \ldots, p_k) \qquad A_i <: A \text{ for } i{=}1, 2 \qquad \mathit{\Gamma}, x_h{:}B, x_t{:}L^{\lhd(\vec{p})}(B) \;{}^{k}{\vdash}^{s_c}_{s'_c}\; e_2{:}A_2\end{array}}{\mathit{\Gamma}, x{:}L^{\vec{p}}(B) \;{}^{k}{\vdash}^{q}_{q'}\; \mathit{match}\; x \;\mathit{with}\; \mid \mathit{nil} \to e_1 \mid \mathit{cons}(x_h, x_t) \to e_2 : A}\;\textsc{T:MatL}$$

Fig. 3. Algorithmic type rules.

The introduction of the partial evaluation rules enables us to formulate a stronger soundness theorem than, e.g., in [9]. It states that the bounds derived from an annotated type statement also hold for non-terminating evaluations. Additionally, the new notation that we use in the operational semantics allows for a more concise statement.

Theorem 4 (Soundness). *Let* $\mathcal{H} \vDash \mathcal{V}{:}\mathit{\Gamma}$ *and* $\mathit{\Gamma} \;{}^{k}{\vdash}^{q}_{q'}\; e{:}A$. *(1) If* $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow v, \mathcal{H}' \mid (p, p')$ *then* $p \leq \mathit{\Phi}_{\mathcal{V},\mathcal{H}}(\mathit{\Gamma}) + q$ *and* $p - p' \leq \mathit{\Phi}_{\mathcal{V},\mathcal{H}}(\mathit{\Gamma}) + q - (\mathit{\Phi}_{\mathcal{H}'}(v{:}A) + q')$.
(2) If $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow \mid p$ *then* $p \leq \mathit{\Phi}_{\mathcal{V},\mathcal{H}}(\mathit{\Gamma}) + q$.

It follows from Thm. 4 and Thm. 2 that run-time bounds also prove termination.

Corollary 1. *Let the resource constants be instantiated by $K^x = 1$, $K^x_1 = 1$ and $K^x_m = 0$ for all x and all $m > 1$. If $\mathcal{H} \vDash \mathcal{V}{:}\Gamma$ and $\Gamma \; {}^{k}\!\vdash^{q}_{q'} \; e{:}A$ then there is an $n \in \mathbb{N}, n \leq \Phi_{\mathcal{V},\mathcal{H}}(\Gamma) + q$ such that $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow v, \mathcal{H}' \mid (n, 0)$.*

Thm. 4 is proved by induction on the derivation of the evaluation statements $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow v, \mathcal{H}' \mid (p, p')$ and $\mathcal{V}, \mathcal{H} \vdash e \rightsquigarrow \mid p$, respectively. There is one proof for all possible instantiations of the resource constants. It is technically involved but conceptually unsurprising. Compared to earlier works [7, 9], further complexity arises from the matching of the constraints in the type rules with the monoid elements in the semantics. The proof can be found in the extended version of this paper.

6 The Inference Algorithm

The inference algorithm is mainly defined by the type rules in the previous section. It works like a standard type inference in which each type is annotated with resource variables and the corresponding linear constraints are collected as each type rule is applied. The main innovation in comparison to the classic algorithm [1] is the resource-polymorphic recursion enabled by the rule T:FUNAPP.

The number of computed constraints grows linearly in the maximal degree k that has to be provided by the user. There is a trade-off between the quality of the analysis and the size of the constraint system. The reason is that one sometimes has to analyze function applications context-sensitively with respect to the call stack. Recall, e.g., the expression *attach(x,attach(y,xs))* from §1 where we used two different types for *attach*.

In our implementation we collapse the cycles in the call graph and analyze each function once for every path in the resulting graph. In a nutshell, the algorithm computes inequalities for annotations of degree k for a strongly connected component (SCC) F of the call graph as follows.

1. Annotate the signature of each function $f \in F$ with fresh resource variables.
2. Use the type rules from §5 to type the corresponding expressions e_f. Introduce fresh resource variables for each type annotation in the derivation and collect the corresponding inequalities.
   (a) For a function application $g \in F$: if $k = 1$ or in the cost-free case use the function resource-monomorphically with the signature from (1). Otherwise, go to (1) and derive a cost-free typing of e_g with a fresh signature. Store the arising inequalities and use the resource variables from the obtained typing together with the signature from (1) in T:FUNAPP.
   (b) For a function application $g \notin F$: repeat the algorithm for the SSC of g. Store the arising inequalities and use the obtained annotated type of g.

The context sensitivity can lead to an exponential blow up of the constraint system if there is a sequence of function $f_1, \ldots, f_n$ such that f_i calls f_{i+1} several times. But such sequences are short in most programs. It would not be a substantial limitation in practice to restrict oneself to programs that feature a collapsed call graph with a fixed maximal path length to obtain a constraint system that is linear in the program size.

In general, the computed constraint systems are simple and can be quickly solved by standard LP-solvers. The objective function states that annotations of arguments in function signatures have to be minimized and that annotations of high degree are more expensive then annotations of low degree.

In the extended version one finds a comparison of the computed evaluation-step bounds with the actual worst-case time behavior for several example programs together with the run times of the analyses. The inference algorithm works efficiently and infers resource-polymorphic types for all programs that we manually typed in our system. However, it is not complete with respect to full resource-polymorphism. This would mean to start with a (possibly infinite) set of annotated function types for each function and to justify each type with a derivation that uses first-order types from the initial set.

The extended version of the paper contains a somewhat artificial example that admits a resource-polymorphic type derivation that cannot be inferred by our algorithm. It seems to be unlikely that there is a method to infer a typing for such functions with a method that uses only linear constraints. One could move to quadratic constraints to address the problem but the efficiency of such an approach is unclear. We plan to also experiment with SMT solvers to deal which such constraints.

7 Related Work

Most closely related is the previous work on automatic amortized analysis [9, 1, 3–5, 7] (see §1). This paper focuses on polymorphic recursion and is the first that investigates relations of the inferred bounds to non-terminating computations.

Other resource analyses that can in principle obtain polynomial bounds are approaches based on recurrences pioneered by Grobauer [12] and Flajolet [13]. In those systems, an a priori unknown resource bounding function is introduced for each function in the code; by a straightforward intraprocedural analysis a set of recurrence equations or inequations for these functions is then derived. A type-based extraction of such recurrences has been given in [14]. Even for relatively simple programs the resulting recurrences are quite complicated and difficult to solve with standard methods. In the COSTA project [15, 16] progress has been made with the solution of those recurrences. In an automatic complexity analysis for higher-order Nuprl terms Benzinger uses Mathematica to solve the generated recurrence equations [17]. Still, we find that amortization yields better results in cases where resource usage of intermediate functions depends on factors other than input size, e.g., sizes of partitions in quick sort. Also compositions of functions seem to be better dealt with by amortization.

A successful method to estimate time bounds for C++ procedures with loops and recursion was recently developed by Gulwani et al. [18, 19] in the SPEED project. They annotate programs with counters and use automatic invariant discovery between their values using off-the-shelf program analysis tools which are based on abstract interpretation. A recent innovation for non-recursive programs is the combination of disjunctive invariant generation via abstract interpretation with proof rules that employ SMT-solvers [20]. In contrast to our method, these techniques can not fully automatically analyze iterations over data structures. Instead, the user needs to define numerical “quantitative functions”. A methodological difference is that we infer (using linear programming) an abstract potential function which indirectly yields a resource-bounding function. The potential-based approach may be favorable in the presence of compositions and data scattered over different locations (partitions in quick sort). Moreover, our method infers tight bounds for functions like insertion sort that admit a worst-case time usage of the form $\sum_{1 \leq i \leq n} i$. In contrast, [18] indicates that a nested loop on $1 \leq i \leq n$ and $1 \leq j \leq i$ is over-approximated with the bound n^2.

The examples from loc. cit. suggest that the two approaches are complementary in the sense that the method of Gulwani et al. works well for programs with little or no recursion but integrate interaction of linear arithmetic with loops. Our method, on the other hand, does not model the interaction of integer arithmetic with resource usage, but is particularly good for analyzing recursive programs involving inductive data types. Moreover, type derivations can be seen as certificates and can be automatically translated into formalized proofs in program logic [21].

Another related approach is the use of sized types [22–24] which provide a general framework to represent the size of the data in its type. Sized types are a very important concept and we also employ them indirectly. Our method adds a certain amount of data dependency and dispenses with the explicit manipulation of symbolic expressions in favour of numerical potential annotations.

Polynomial resource bounds have also been studied in [25] that addresses the derivation of polynomial size bounds for functions whose exact growth rate is polynomial.

8 Conclusion and Future Research

We have continued our work on automatic type-base amortized analysis for polynomial resource bounds. To deal with the challenge of resource-polymorphic recursion we have introduced a new inference algorithm. It uses a special cost-free resource metric to compute alternate function types for recursive calls. The algorithm has been implemented and it has been shown by experiments that it efficiently computes types for interesting examples such as sorting algorithms. To prove the non-trivial soundness of the algorithm for terminating and non-terminating evaluations we introduced a novel partial big-step operational semantics. It models non-termination with non-deterministic inductive rules.

Even though there are examples that the inference algorithm cannot handle we find it to be a good compromise between efficiency and performance. Therefore, our future research will focus mainly on conceptual extensions of the type system that will employ the same inference method. Most notably we plan an extension to mixed potential capable of inferring bounds like $n \cdot m$, an extension to recursion on non-inductive data like integers, and the integration of higher-order and polymorphism.

References

1. Hofmann, M., Jost, S.: Static Prediction of Heap Space Usage for First-Order Functional Programs. In: 30th ACM Symp. on Principles of Prog. Langs. (POPL'03). (2003) 185–197
2. Tarjan, R.E.: Amortized Computational Complexity. SIAM J. Algebraic Discrete Methods **6**(2) (1985) 306–318

3. Hofmann, M., Jost, S.: Type-Based Amortised Heap-Space Analysis. In: Prog. Langs. and Systems, 15th European Symp. on Prog. (ESOP'06). (2006) 22–37
4. Hofmann, M., Rodriguez, D.: Efficient Type-Checking for Amortised Heap-Space Analysis. In: 18th Conf. on Comp. Science Logic (CSL'09), LNCS (2009)
5. Jost, S., Loidl, H.W., Hammond, K., Scaife, N., Hofmann, M.: Carbon Credits for Resource-Bounded Computations using Amortised Analysis. In: 16th Intl. Symp. on Form. Meth. (FM'09). (2009) 354–369
6. Campbell, B.: Amortised Memory Analysis using the Depth of Data Structures. In: 18th Euro. Symp. on Prog. (ESOP'09). (2009) 190–204
7. Jost, S., Hammond, K., Loidl, H.W., Hofmann, M.: Static Determination of Quantitative Resource Usage for Higher-Order Programs. In: 37th ACM Symp. on Principles of Prog. Langs. (POPL'10). (2010) 223–236
8. Atkey, R.: Amortised Resource Analysis with Separation Logic. In: 19th Euro. Symp. on Prog. (ESOP'10). (2010) 85–103
9. Hoffmann, J., Hofmann, M.: Amortized Resource Analysis with Polynomial Potential. In: 19th Euro. Symp. on Prog. (ESOP'10). (2010) 287–306
10. Cousot, P., Cousot, R.: Inductive Definitions, Semantics and Abstract Interpretations. In: 19th ACM Symp. on Principles of Prog. Langs. (POPL '92). (1992) 83–94
11. Leroy, X.: Coinductive Big-Step Operational Semantics. In: 15th Euro. Symp. on Prog. (ESOP'06). (2006) 54–68
12. Grobauer, B.: Cost Recurrences for DML Programs. In: 6th Intl. Conf. on Funct. Prog. (ICFP'01). (2001) 253–264
13. Flajolet, P., Salvy, B., Zimmermann, P.: Automatic Average-case Analysis of Algorithms. Theoret. Comput. Sci. **79**(1) (1991) 37–109
14. Crary, K., Weirich, S.: Resource Bound Certification. In: 27th ACM Symp. on Principles of Prog. Langs. (POPL'00). (2000) 184–198
15. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost Analysis of Java Bytecode. In: 16th Euro. Symp. on Prog. (ESOP'07). (2007) 157–172
16. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis. In: 15th Symp. Stat. An. (SAS'08). (2008) 221–237
17. Benzinger, R.: Automated Higher-Order Complexity Analysis. Theor. Comput. Sci. **318**(1-2) (2004) 79–103
18. Gulwani, S., Mehra, K.K., Chilimbi, T.M.: SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In: 36th ACM Symp. on Principles of Prog. Langs. (POPL'09). (2009) 127–139
19. Gulavani, B.S., Gulwani, S.: A Numerical Abstract Domain Based on Expression Abstraction and Max Operator with Application in Timing Analysis. In: Comp. Aid. Verification, 20th Int. Conf. (CAV '08). (2008) 370–384
20. Gulwani, S., Zuleger, F.: The Reachability-Bound Problem. In: Conf. on Prog. Lang. Design and Impl. (PLDI'10). (2010) 292–304
21. Beringer, L., Hofmann, M., Momigliano, A., Shkaravska, O.: Automatic Certification of Heap Consumption. In: Log. f. Prog., AI, and Reas., 11th Conf. (LPAR'04). (2004) 347–362
22. Hughes, J., Pareto, L., Sabry, A.: Proving the Correctness of Reactive Systems Using Sized Types. In: Symp. Princ. of Prog. Langs. (POPL'96). (1996) 410–423
23. Hughes, J., Pareto, L.: Recursion and Dynamic Data-structures in Bounded Space: Towards Embedded ML Programming. In: 4th Intl. Conf. on Funct. Prog. (ICFP'99). (1999) 70–81
24. Chin, W.N., Khoo, S.C.: Calculating Sized Types. High.-Ord. and Symb. Comp. **14**(2-3) (2001) 261–300
25. Shkaravska, O., van Kesteren, R., van Eekelen, M.C.: Polynomial Size Analysis of First-Order Functions. In: Typed Lambda Calc. Apps. (TLCA'07). (2007) 351–365